ANALYSIS OF ALGORITHMS - INTRODUCTION

By Elias Debelo

JANUARY 1, 2022
HARAMAYA UNIVERSITY

CHAPTER ONE

INTRODUCTION

1.1. What is an Algorithm?

- ✓ An algorithm is a finite set of instructions that, if followed, accomplishes a particular task.
- ✓ Algorithms is a sequence of computational step that transform input into output
- ✓ We can also view an algorithm as a tool for saving a well-specified computational problem.

The statement of the problem specifies in general terms the desired input/output (I/O) relationships. The algorithm describes a specific computational procedure for achieving that I/O relation.

> *E.g., One might need to sort a sequence of numbers into nondecreasing order. This problem arises frequently in practice and provides fertile ground for introducing standard design techniques and analysis.*

Then, what kinds of problems are solved by algorithms?

Practical applications of algorithms are ubiquitous and includes the following examples;

- ✓ Human Genome Project has a goal of identifying all the 100,100 genes in human DNA, determining the sequence of 3 billion chemical base pairs that makeup human DNA, storing information in database, and developing tools for data analysis. Each these steps require sophisticated algorithms.
- ✓ The internet enables people all around the world to quickly access and retrieve large amount of information. This also need a cleaver algorithm.
- ✓ Electronic commerce enables goods and services to be negotiated and exchanged electronically. This needs a number of security mechanisms and algorithms/ Thus, we need cryptography and digital signature for encrypting and signing our information. And many more.

Besides, all algorithms satisfy the following criteria:

1. **Input** – Zero or more quantity are externally supplied.
2. **Output** – At least one quantity is produced.

3. **Definiteness** – Each instruction is clear and unambiguous.
4. **Finiteness** – If we trace out the instructions of an algorithm, then for all cases, the algorithm terminates after a finite number of steps.
5. **Effectiveness** – Every instruction must be very basic so that it can be carried out, in principle, by a person using only pencil and pen.

An algorithm that is definite and effective is called computational procedure. Then, a program is the expression of an algorithm in programming language.

1.2. Algorithm Specification

We can describe an algorithm in many ways. Natural language like English, graphical representation like flowcharts or using pseudocodes.

The Pseudocode Convention

1. Comments begin with //and continues until the end of line
2. Block are indicated with braces: {and}, and the statements are delimited by;
3. An identifier begins with a letter.
4. Assignment of values to variables is done using the assignment statement. <variable> := <expression>
5. There are two Boolean values **true**/**false**. In order to produce these values, the logical operators **and**, **or** and **not** and the relational operators <, >, =, ≠, <=, and >= are provided.
6. Elements of multidimensional arrays are accessed using **[** and **]**. E.g. If A is two dimensional array, the (i, j)th element of the array is denoted as A[I, j]. and array indices start at zero.
7. The following looping statements are employed: for, while, and repeat-until.

   * **while** <condition> **do**

   {

   <statement 1>
   .
   .
   <statement n>

   ~ As long as <condition> is **true**, the statement get executed.

   ~ When <condition> becomes **false**, the loop is exited.

   ~ The value of <condition> evaluated at the top of the loop.

   }

* **for** variable:=value1 **to** value2 **step** step **do**
{
<statement 1>
.
.
<statement n>
}

~ The value1, value2, and step are arithmetic expressions.

~ The clause '**step** step' is optional, and it takes +1 if it does not occur. Step could be positive or negative, is tested for termination at the start of each iteration.

* **repeat**

<statement 1>
.
.
<statement n>
Until <condition>

~ The statement executed as long as <condition> is **false**.
~ The value of <condition> is computed after executing the statement.

The instruction **break**; can be used within any of the above loops. The **return**, instruction also used in the above loops for existing like **break**. In addition, it also exits the function itself.

8. Condition statements
if <condition> **then** <statement>
if <condition> **then** <statement 1> **else** <statement 2>

<condition> is a Boolean expression.

9. Cases
Case
{
:<condition 1>: <statement 1>
.
.
:<condition n>: <statement n>
:**else:**
<statement n+1>
}

if <condition 1> is **true**, <statement 1> gets executed and the case exited.
if none of the above <conditions> are **true**, <statement n+1> is executed and **case** statement exited.
the **else,** is optional.

10. The input/output are done using **read** and **write.**
11. An algorithm consists of head and body. The head takes the following form:

```
Algorithm Name (<Parameter List>)
```

E.g.

```
Algorithm Max (A, n)
{
      //A is an array of size n
      Result:= A[1];
      for i:=2 to n do
            if A[i] > Result
                  then Result:= A[i];
      return Result;
}
```

In the above algorithm (named Max), A and n are procedure (function, method) parameters. Result and i are local variables. The algorithm returns the largest array element.

1.3. Recursive Algorithms

- ✓ A recursive function is a function that define in terms of itself.
- ✓ Similarly, an algorithm is said to be recursive if the same algorithm is invoked in the body. And an algorithm that calls itself is *direct recursive*.
- ✓ Algorithm A is said to be *indirect recursive* if it calls another algorithm which in turn calls A.

E.g., Let say we have factorial problem, it can be solved by looping, but recursive way is much easy and fair to understand.

Factorial of 5 is 5! Which is equal to 5*4*3*2*1 = 120

So, using recursive function

```
Algorithm Fact(n)
{
      if n <= 1             //base case
            then return 1;
      else
            return n*Fact(n-1);
}
```

~ A function Fact is calling itself, this phenomenon is called recursion, and the function containing recursion is called recursive function.

~ In the recursive program, the solution to the base case is provided and the solution of the bigger problem is expressed in terms of smaller problems.

The execution goes like,

```
Fact(5)
     5*Fact(4)
          4*Fact(3)                    ~ then the results are
               3*Fact(2)                 collect all together.
                    2*Fact(1)
                         Return 1
```

1.4. Algorithm Analysis

- ✓ This refers to the process of determining the amount of computing time and space required by different algorithms.
- ✓ In other words, it is the process of predicting the resource requirement of algorithm in a given environment.

* Computing time and space requirements are the two most important criteria for judging algorithm bad or good, this two have direct relation with performance.

Complexity Analysis is concerned with determining the efficiency of algorithms.

* *Space Complexity* of an algorithm is the amount of memory it needs to run to completion.
* *Time Complexity* of an algorithm is the amount of computer time it needs to run to completion.

Performance evaluation can be loosely divided into two phases: (1) a priori estimates – performance analysis and (2) a posteriori testing – performance measurement.

For analyzing and algorithm we need model of implementation technology that will be used. Let assume Random Access Machine (RAM) model of computation. In the RAM model, instructions are executed one after the other, with no concurrent operations.

* Then we need to define the instructions of the RAM model and their costs. Where each instruction takes a constant amount of time.

NB: The time taken by an algorithm grows with the size of the input.

Analysis of Insertion Sort

So, it is traditional to describe the running time of a program as the function if the size of its input.

- ✓ *Input Size* depends on the type of problem being studied. For sorting it is the number of items in the input. Say 'array size of n.'
- ✓ *Running Time* of an algorithm is number of primitive operations on "steps" executed.

* A constant amount of time is required to execute each line of our pseudocodes. One line may take different amount of time than the other line.

Now, the insertion sort procedure with the time "cost" of each statement and number of times each statement is executed.

Algorithm InsertionSort(A, n)	Cost	Times
{		
for j:=2 to n do	c1	n
key:=A[j];	c2	n-1
i:=j-1;	c3	n-1
while i>0 and A[i] > key do	c4	$\sum_{j=2}^{n} t_j$
A[i+1]:=A[i]	c5	$\sum_{j=2}^{n} (t_j\text{-}1)$
i:i-1;	c6	$\sum_{j=2}^{n} (t_j - 1)$
A[i+1]:=key;	c7	n-1
}		

* Where, A is the array of size n,
t_j is a number of times the while loop test is executed for the values of j.

Therefore, the running time of the algorithm is: The sum of running times for each statement executed. Statement that takes Ci to execute and is executed n times will contribute **Cin** to the total running time.
Compute the total running time T(n), the sum of product of the cost and times column.

Therefore,

$T(n)=c1n+c2(n-1) + c3(n-1) + c4\sum_{j=2}^{n} t_j + c5\sum_{j=2}^{n}(t_j -1) + c6\sum_{j=2}^{n}(t_j - 1) + c7(n-1)$

$T(n) = n(c1+c2+c3+c7) - (c2+c3+c7) + c4\sum_{j=2}^{n} t_j + c5\sum_{j=2}^{n}(t_j -1) + c6\sum_{j=2}^{n}(t_j - 1)$

* ***An algorithm analysis should be categorized in some way***. Algorithm must be examined under different situations to correctly determine their efficiency for accurate comparison.

~ Best Case Analysis: Assumes that data are arranged in the most advantageous order.

* It also assumes the minimum input size.
* Computes the lower boundary of T(n).
* It causes fewest number of executions.

E.g., Sorting – the best case happens if the data are arranged in the required order. For Searching, it happens when the required item found at the beginning.

~ Worst Case Analysis: Assumes that the data are arranged in a disadvantageous order.

* It assumes larger input size.
* It computes the upper bound of T(n) and causes maximum number of executions.

E.g., Sorting – Data are arranged in opposite to the required order. For searching, the required item found at the end or item missing.

~ Best Case Analysis: Assumes the data are found in a random order.

* It also assumes random or average input size.
* It computes optimal bound of T(n) and causes average number of executions.

E.g., Sorting – data are in random order. For searching, the required item found at any position or missing.

* Best case and average cases only cannot be used to determine or estimate complexity if an algorithm.
* Worst case is the best to determine the complexity, because,
  - It is the longest running time for any input of size n, which give us guarantee that the algorithm will never take any longer.
  - It occurs fairly often.
  - Much of the time average case fall to worst case.

Order of Magnitude

- ✓ It refers to the rate at which the storage/time grows as a function of problem size.
- ✓ It expressed in terms of its relationship to some known functions – asymptotic analysis.

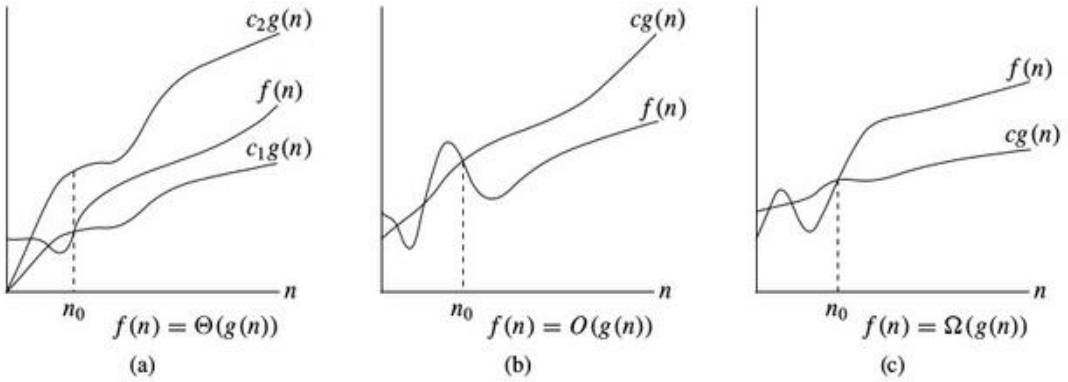
- ✓ Can be called *rate of growth,* or *order of growth*

Based on *asymptotic analysis* – giving limits and *asymptotic tight* – getting an appropriate upper/lower bound or both bounds.

Big-O notation*(O)* – Upper bound to T(n), represent a worst-case analysis.

Big-Omega (Ω) – Lower bound to T(n), represents best case analysis.

Big-Theta (Θ) – Upper and lower bound to T(n), describes the average case analysis.



* Back to insertion sort,

The best case occurs if the array is already sorted. For each j=2, 3..., n, we then find that A[j]<=key when i has its initial value j-1. Thus, t_j=1 for j=2, 3...n, and the best-case running time is:

T(n) = n(c1+c2+c3+c7) – (c2+c3+c7) + c4$\sum_{j=2}^{n} 1$ + c5$\sum_{j=2}^{n}(1-1)$ + c6$\sum_{j=2}^{n}(1-1)$

T(n) = n(c1+c2+c3+c7) – (c2+c3+c7) + c4n + 0 + 0

T(n) = n(c1+c2+c3+c4+c7) – (c2+c3+c7)

T(n) = an + b, where, a represent c1+c2+c3+c4+c7, and b replaces the other part c2+c3+c7. No confusion that, the plus/minus sign won't change the order of magnitude.

The running time is then linear function of n, for a and b depends on the constant statement costs.

Now, if the array is in reverse sorted order against the required order – sorting result in worst case. Then we must compare each element A[j] with each element in the entire sorted subarray A[1:j-1] or A[1, 2, 3 …n], and so t_j=j, for j=2, 3, … n.

Therefore, the running time of insertion sort is:

$T(n) = n(c1+c2+c3+c7) - (c2+c3+c7) + c4(\frac{n(n+1)}{2} - 1) + c5(\frac{n(n-1)}{2}) + c6(\frac{n(n-1)}{2})$

Using summation definition, from arithmetic series

$$\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1 \qquad \text{and } \sum_{j=2}^{n}(j-1) = \frac{n(n-1)}{2}$$

After multiplications and rearranging; **$T(n) = an^2 + bn + c$**, which is quadratic function of n.

To describe using order of magnitude of insertion sort; best case, Ω(n) and worst case, $o(n^2)$. Average case analysis falls to worst case. And the space complexity of insertion sort is stable with O(1). The reason is that, insertion sort is in-place sort.

NB: bn and c are irrelevant compared to an2 as n gets larger. Likewise, the coefficient will have less effect than the exponent. This means, the leading term's exponent dominates execution time the algorithm as the size of n grow! So, we drop them.

1.5. Elementary Data Structures

Stacks and Queues

- ✓ Stacks and queues are dynamic sets in which the element removed from the set by the delete operation is prespecified.

* *Stacks*

Implements a last-in, first-out or LIFO policy. Insertion operation on a stack is often called PUSH, and delete operation, which does not take an element argument, is often called POP.

Array has an attribute top[S], that indexes the most recently inserted element. The stack consists of elements S[1 … top[S]], where S[1] is element at the bottom of the stack and S[top[S]] is element at the top. Where, S is an array of size n.

* **Algorithm** StackEmpty(S)

```
{
      if top[S]=0 then return True;
      else return false;
}
```

* **Algorithm** Push(S, x)
  {
  top[S]:=top[S] + 1;
  S[top[S]]:=x;
  }

* **Algorithm** Pop()
  {
  if StackEmpty(S) then **error** “Underflow”;
  else top[S]:=top[S] – 1;
  return S[top[S]+1];
  }

Now, it is your turn ***to check if the stack is full before pushing an element into the stack by modifying the above algorithm***. Can you try on paper?

- Each of the three stack operation takes O(1) time. Because all statements execute at constant time, ci.

* *Queues*

Implements first-in first-out, or FIFO policy. Insert operation is called ENQUEUE, and delete operation is called DEQUEUE. Unlike stack, queue has *head* and *tail*.

When an elements enqueued it takes place at the tail of the queue, but element dequeued are always the one at the head of the queue.

Unlike stacks queues can be circular, meaning if queue get full while there are spaces, enqueueing will insert new element at the position indicated by tail[Q]. The following example we implement queue of element from 1 to n-1, using array of n size. So, it can only contain n-1 elements;

1	2	3	4	5	6	7	8	9	10	11	12
						15	6	9	8	4	

head[Q] = 7, Q[head[Q]] = 15, tail[Q] = 12

1	2	3	4	5	6	7	8	9	10	11	12
3	5					15	6	9	8	4	17

head[Q] = 7, Q[head[Q]] = 15, tail[Q] = 3

1	2	3	4	5	6	7	8	9	10	11	12
3	5					~~15~~	6	9	8	4	17

head[Q] = 8, Q[head[Q]] = 6, tail[Q] = 3

A queue Q, above is a circular queue, where tail[Q] holds index of next place new element will be inserted. Numbers ranging from 1 to 12 are indexes of the array.

The elements in the queues are at locations head[Q], head[Q]+1, ... tail[Q]-1, where *'wrap around',* in the sense that location 1 immediately follows location n in a circular order.

* When head[Q]=tail[Q], the queue is empty, but initially head[Q]=tail[Q]=1
* When head[Q]=tail[Q]+1, the queue is full.

```
Algorithm Enqueue(Q, x)
{
   Q[tail[Q]]:=x;
   if tail[Q]=length[Q] then tail[Q]:=1;
   else tail[Q]:=tail[Q] + 1;
}
```

```
Algorithm Dequeue(Q)
{
      x:=Q[head[Q]];
      if head[Q]=length[Q] then head[Q]:=1;
      else head[Q]:= head[Q] + 1;
      return x;
}
```

- Each queue operation takes O(1) time.

Now, ***how can you modify this algorithm to create a queue that can hold n element, unlike the above n-1?***

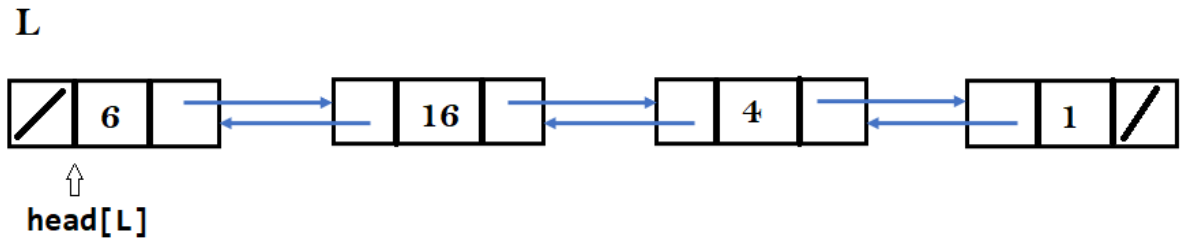
* *Linked Lists*

- ✓ Linked list (LL) is data structure in which the objects are arranged in a linear order.
- ✓ Unlike an array, though, in which the linear order is determined by array indices, the order in a LL is determined by a pointer in each object.
- ✓ LL can be singe LL, doubly LL or circular LL.

* Doubly LL is LL having two pointers, prev and next besides key or field.

Double linked list L, given an element x in the list, next[x] is points to its successor in the linked list, and prev[x] points to its predecessor.

If prev[x] is NIL, the element x has no predecessor and is therefore the first element, or **head**, of the list. If next[x] is NIL, the element x has no successor and is therefore the last element, or **tail**, of the list.



Head[L] – head of the list, L. next of head is node containing value of 16 and prev of head is NIL, represented by /.

Searching in LL, n is number of nodes in the list L.

Algorithm ListSearch(L, x)	Cost	Times	Weight
{			
k:=head[L];	c1	1	c1
while x≠NIL and key[x]≠k do	c2	n	c2n
x:=next[x];	c3	n-1	c3n-c3
return x;	c4	1	c4
}			

T(n) = n(c2 + c3) + c1 – c3 + c4, since ci are constants, we can represent them as a or b.
T(n) = an + b, running time of searching node in the list is linear. It takes O(n) time.

Inserting and deleting nodes, left for you.

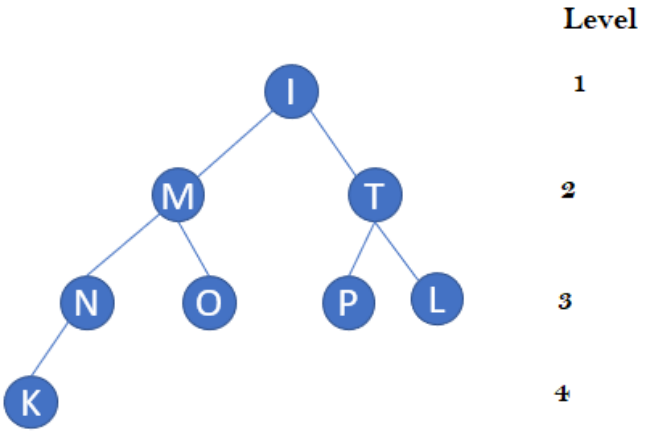
* Trees

- ✓ A tree is a finite set of one or more nodes such that there is specially designated node the root and the remaining nodes are partitioned into n>=0 disjoint sets T1, T2, ..., Tn, where each of

these sets is a tree. The sets T1, T2, ... Tn is called subtree of the root.

- ✓ Tree is also called *connected graph* without cycle, where connected graph is a graph with no independent vertex (or say node for tree).
- ✓ Tree is an abstract model of hierarchical structures.

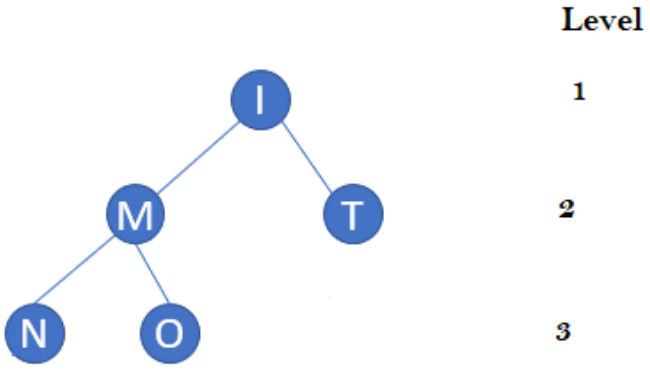
* Each node can have a number of child nodes, and the parent and child nodes are connected by arcs.
* On the tree below; node I is parent of nodes M and T.



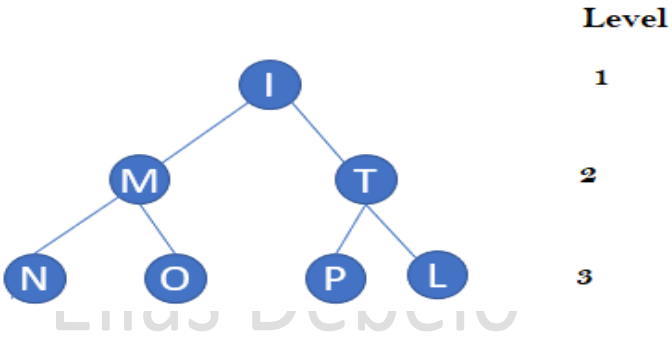
* Nodes K, O, P, and L are called leaves or terminal nodes and the other nodes are nonterminal or internal nodes.
* Nodes P and L, M and T, N and O are called siblings.
* A path through the tree to a node is a sequence of arcs that connects the root to the node.
* The length of a path is a number of arcs in the path.
* The level of a node is equal to the length of the path from the root to the node plus one.
* The height of a tree is equal to the maximum level of a node in the tree. E.g., the height of the above tree is 4.

Binary Tree (BT)

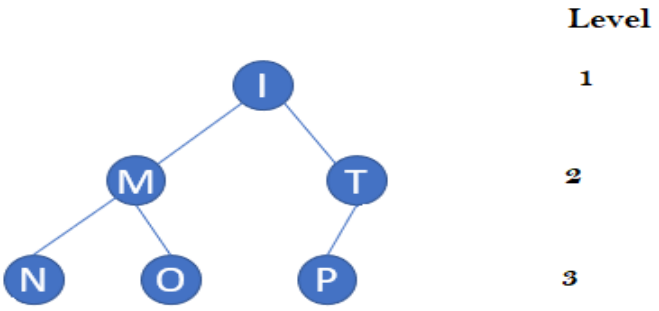
- ✓ BT is a tree whose nodes have at most two children. Meaning that, nodes can have one child (list or right) or no child.
- * **Full BT**
  - A binary tree where each node has either zero or two children.
  - It cannot have single child.



* **Perfect BT**
  - BT in which every internal node has exactly two child node and all the leaf nodes are at the same level.

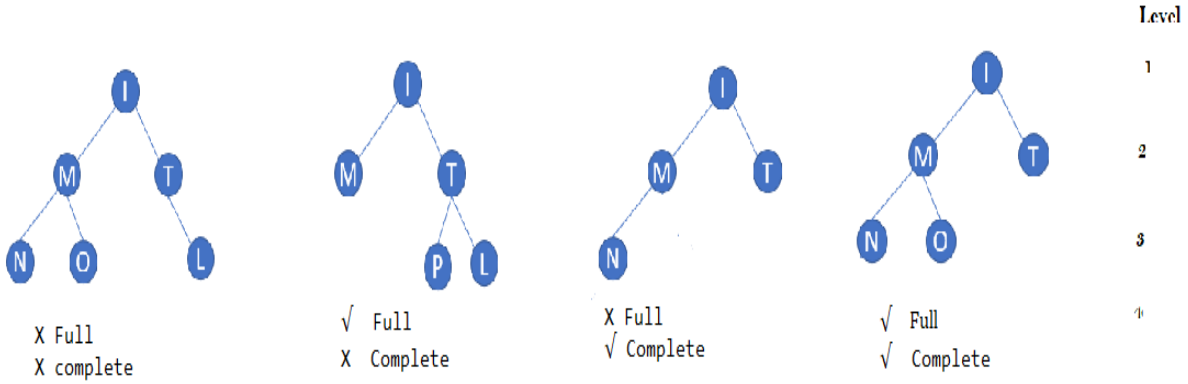


* **Complete BT**
  - BT just like full BT, but with two major differences:



1. All the leaf elements must lean towards the left.
2. The last leaf element might not have a right sibling.

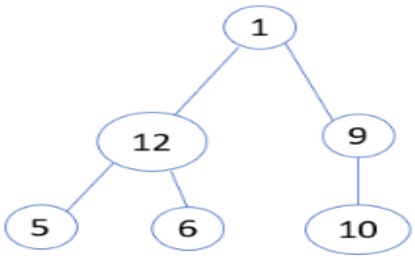
Comparisons between *full* and *complete* BTs:



How to create Complete BT from an array?

1	12	9	5	6	10

1. Set the first element root node.
2. Put second and third element as left and right child of the root respectively.
3. Put the next two element as left and right children of the left most node.
4. Keep repeating until the last element.



* Binary Search Trees (BST)

- ✓ It is BT, it may be empty, if not it should satisfy the following conditions.
  - Every element has a key, and no two elements have the same key (i.e., the keys are distinct)
  - The keys in the left subtree are smaller than the key in the root.
  - The keys in the right subtree are greater that the key in the root.
  - The left and right subtrees are also BSTs.

E.g.,

```
Algorithm SearchBST(t, x)                                  Costs      Times
{
     if t = NIL then                                       c1         1
          return NILL;                                     c2         1
     else
          if x = t->data then                              c3         1
               return x;                                   c4         1
          else
               if x < t->data then                        c5         1
                    return SearchBST(t-lchild, x);        c6         h/2
               else return SearchBST(t->rchild, x);       c7         h/2
}
```

* t->data means a way of accessing data/value pointed by node t. that is using node name (t) as pointer.
* t->lchild – left child of t and t-rchild – right child of t.

Besides searching for a key stored in the tree, BST can support such queries; MINIMUM, MAXIMUM, SUCCESSOR, and PREDECESSOR of certain node.

Analysis

The running time of BST takes O(h), where h is the height of the tree. Therefore, all operations from 1 to 5 takes constant times 1. The recursive calls at line 6 and 7 takes h/2. Because the recursive calls move left and right.

T(n) = c6(h/2) + c7(h/2) + c1+c2+c3+c4+c5
T(n) = ½ (c6 + c7) + b
T(n) = ½ ah + b, that is linear time. Searching time correspond to the height of the tree.

⇨ O(h), in worst case.

Let us search a node that contain the smallest value of key in the tree. How? Just move to the left most leaf.

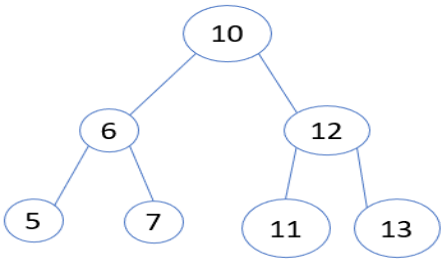
```
Algorithm Min(t)
{
     x:=root(t);                  ~ this algorithm returns node
     while left[x] ≠ NIL do       containing the smallest value.
          x:=left[x];             NB: even if the tree does not have left
     return x;                    subtree, key[x] is still the smallest,
```

}

because by definition of BST. Every right node is bigger than its root.

* *Hand on paper. How to fin maximum key?* ***Left for you.***

Tree Traversal: or tree walk, that is visiting every node on the tree in some predefined orders; preorder, inorder, and postorder.
Look at the following tree and observe the result as follow;
* Preorder – visit the root node fist, then left subtree, and right subtree. **Result**: 10 6 5 7 12 11 13
* Inorder – Walk starting from the left, visit root then visit right. **Result**: 5 6 7 10 11 12 13
* Postorder – Traverse left subtree, then right subtree, lastly visit the root. **Result**: 5 7 6 11 13 12 10



1.6. Priority Queues

- ✓ Any data structure that supports the operations of search min (or max), insert, and delete min (or max) is called, priority queues.
- ✓ It also called generalization of queues and stacks.

~ it has various areas oof applications like job scheduling a computer system, where, the keys might correspond to the 'priorities' which indicate which user or process should be processed first.

E.g., Selling services of the machine.

"Our aim is to maximize the return or gain from the machine."

* Let's say every user pay a fixed amount per use, however, time used by each user is different. Assume that there is a list of waiting users in the queue (list). So, to hit our aim we need to select a user with smaller time use. (This require removing min from the queue, means send user to use the machine).

* But if a user has a fixed time use and pay different amount per use, we need to select user that pay us larger from the queue. (That is removing max from queue and sending to machine for use).

***** is have n sized queue where insertion and deletion of max value supported;

- It will take O(1) to insert an element into unsorted queue, and O(n), for finding and removing max element.
- It will take O(n) to insert an element into queue of sorted elements in nondecreasing order. But It take O(1) for deleting max element from the queue.

Now, using max-heap additions and deletions can be performed in O(nlogn) time.

1.7. Heaps

* Heap (min/max) is a complete binary tree with the properties that the value at each node is at least as small as (as large as) the value at its children. If exists. This is called the *heap property*. Heap can be called binary heap.

* Inserting item into the queue is very efficient. But finding the largest element needs scanning the whole list. Okay, what if we use simple sorting for that? Well, inserting new node needs us to shift existing list for creating room from upcoming item.

Data structure that supports both is called max-heap. It contains the largest element at the root.

~ An array A that represent a heap is an object with two attributes; length[A], which is the number of elements in the array and heap-size[A], which is the number of elements in the heap stored within array A.

- ✓ The root of the tree is A[1], and given index i of a node; parent of i , left of i, right of i can calculated as:

Algorithm Parent(i)

{ return Li/2⌋; }

Algorithm left(i)

{ return 2i; }

Algorithm right(i)

{ return 2i+1; }

In a max-heap *A[parent(i)] >= A[i]* and for min-heap *A[parent(i)] <= A[i]*.

- **Maintaining the Heap Property**

* Every node indexed at i in the heap tree is greater or equal with its left and right children.

```
Algorithm Max-Heapify(A, i)
{
     l:=left(i); r:=right(i);
     if l<=heap-size[A] and A[l]>A[i] then
          largest:=l
     else  largest:=i;
     if r<=heap-size[A] and A[r]>A[largest] then
          largest:=r;
     if largest ≠ i then
          exchange A[i] with A[largest];
          Max-Heapify(A, largest);
}
```

* Now, if A[i] is largest, then the subtree rooted at node i is a Max-heap and the algorithm terminate.

* Otherwise, one of the two children has the largest element and A[i] swapped with A[largest], which causes node i and its children to satisfy the max heap property.

* The node indexed by the largest may violet the max heap property since it now holds the value of original A[i]. Consequently, Max-Heapify must be called recursively on that subtree.

- **Building a Heap**

~ We use Max-Heapify in a bottom-up manner to convert an array A[1,...n], where n = length[A] into max-heap.

NB: the element is the subarray A[(n/2 ⌋+1 ... n] are all leaves of the tree, and so each is a 1-element heap to begin with. E.g., if n=7, then all elements starting from index 4 to n are all leaves.

```
Algorithm Build-Max-Heap(A)
{
     Heap-size[A]:=length[A];
     for i:= Llength[A]/2⌋ to 1 step -1
          do Max-Heapify(A,i);
}
```

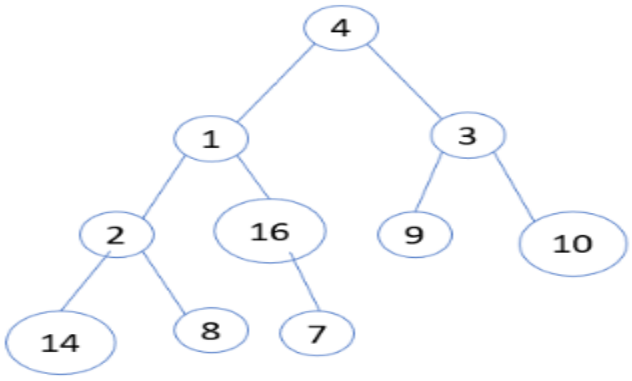
~ Running Max-Heapify on a subtree of size n rooted at a given index, node is the O(1) time to fix-up the relationship among the element A[i], A[left(i) and A[right(i)] plus the time to run Max-Heapify on a subtree rooted at one of the children of i.

~ And each call to Max-Heapify costs O(logn) time, and there are n such calls or O(n). Thus, the running time is O(nlogn). But this upper bound is not asymptotically tight. The tighter upper-bound is O(n) for building max-heap from unordered array.

E.g., Array A,

1	2	3	4	5	6	7	8	9	10
4	1	3	2	16	9	10	14	8	7

Converting into binary heap tree;



Now, find index i to start with, start from non-terminal nodes next to the leaf.

$$i = \lfloor length[A]/2 \rfloor = 10/2 = 5.$$

A[i] = A[5] = 16, call Max-Heapify(A,5).

Left(5) = 7 and right(5) = no right child. Then compare 16 with 7, there is no swapping because 16 > 7.

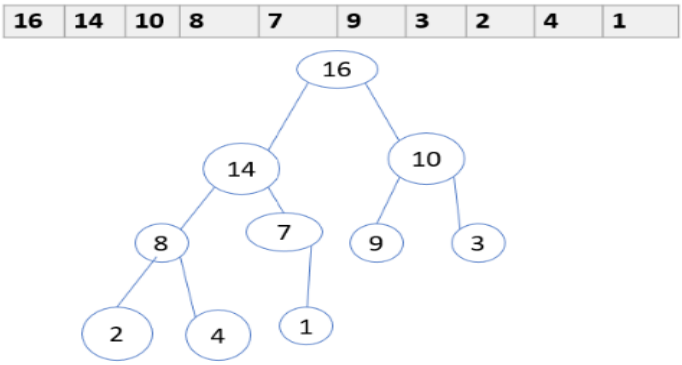
Decrement i to 4:

A[i] = A[4] = 2, left(4)=14 and right(4) = 8. Compare 2 with 14, since root (A[i]) is less than its left child, 14 or A[8]. Largest = index of left, which is 8.

Now compare A[largest] = A[8] = 14 with its sibling, node A[9] = 8. Now, since A[8] > A[9], value of largest which was 8, unchanged.

...

Now, exchange A[4] with A[8]. Keep on doing by calling Build-Max-Heap, the final tree would look like;



The aim was to bring the maximum element at front. **Do it until you reach the at the tree above.**

- **The Heapsort Algorithm**

~ if we exchange A[1] with A[n] to bring the max value at the end. Now, if we "discard" node n from the heap by decrementing heap-size[A], we observe that A[1 ... n-1] can easily be made into max-heap.

~ the children of root remain max-heaps, but the new root may violet max-heap property. All that is needed is to restore the max-heap A[1 ... n-1] is one call to Max-Heapify(A, 1).

```
Algorithm Heapsort(A)                          Cost       Weight
{
     Build-Max-Heap(A);                        c1         c1nlogn
     for i:=length[A] to 2 step -1             c2         c2n
          do exchange A[1] with A[i];          c3         c3(n-1)
          heap-size[A]:=heap-size[A]-1;        c4         c4(n-1)
          Max-Heapify(A, 1);                   c5    c5(n-1(logn))
}
T(n) = c1nlogn+c2n+c3n-c3+c4n-c4+c5nlogn-c5logn
T(n) = (c1+c5) nlogn+n(c2+c3+c4)-c3-c4-c5logn
```
T(n) = nlogn, where other terms are constants and linear funtions.

The heapsort takes O(nlogn) time. Since, the call to Build-Max-Heap(A) takes O(n) and each of n-1 call to Max-Heapify takes time of O(logn).